An Architectural Style for Data-Driven Systems

Reza Mahjourian

Department of Computer and
Information Science and Engineering
University of Florida
Gainesville, FL 32611, USA
rezam@ufl.edu

Abstract

Data-driven systems and applications are very specialized software solutions for acquisition, management, and presentation of information. These systems are usually developed using the same software tools, technologies and processes as any other type of software. Not only is this approach inefficient, but also it results in extreme redundancies due to the inherently repetitive nature of these applications. This is while data-driven systems exhibit characteristics which can be exploited for extensive reuse across a single application or a family of applications. In this paper, we present our experiences over the past six years in constructing and deploying XPage, an architectural style and its accompanied development framework, which are especially designed for data-driven systems. We discuss the challenges and trade-offs involved in designing XPage, as well as the lessons that we learned in the process. Several case studies are presented to demonstrate XPage's efficiency and flexibility for developing real-world solutions.

1 Introduction

Data-driven systems are software solutions for data/information management. The two primary functions of these systems are information acquisition and retrieval. Information acquisition is typically performed via data entry forms or interfacing with external data sources. Information retrieval is concerned with presenting the stored information to the user with appropriate navigation and querying facilities. These systems are also characterized by intensive user interactions both during acquisition and retrieval of information. Data-driven applications are beyond doubt one of the most common types of customized software systems in use today. University registration systems, e-commerce applications, content management systems, financial and accounting applications, a personal address book, and an online photo album are a few examples of data-driven applications.

Despite existence of a consistent demand for new data-driven systems, they are mostly developed as one-off projects, with little reuse taking place beyond what is offered by the development technologies and programming languages used. An observation is support of this fact is that still the de facto standard for developing web-based software solutions, many of which fall under the category of data-driven applications, is *LAMP* (Linux, Apache, MySql, PHP), or one of its variants. It is while data-driven systems exhibit great potential for extensive reuse, even across systems developed in completely unrelated domains. Recently, the software industry has introduced some development frameworks offering higher level programming libraries to help with rapid development of data-driven applications. However, these frameworks are not high-level enough to prevent the repetitive nature of data-driven systems from showing up as nearly identical programming constructs. Moreover, none of these industrial frameworks offer an explicit software architecture and the software engineering decisions behind their designs are just buried in their implementations.

Academic research on producing agile techniques or methodologies for developing data-driven systems is severely lacking as well. Software engineering researchers consider data-driven applications to be in the realm of database research, because of their concentration on information management tasks. Moreover, the seemingly primitive nature of "reading and writing structured data" appears to be lacking the necessary complexity to qualify as an interesting software research problem. On the other hand, database researchers have little interest in solving the software engineering challenges involved in streamlining devel-

opment and managing the complexity of software systems. In spite of that, it is quite surprising to know that the very little work done in this area comes from the database research community and not the software research community.

In this paper, we present an architectural style [11, 7] specifically designed for creating and maintaining data-driven systems. This architectural style has been extracted from a framework developed in 2001, and gradually extended afterwards. We have coined the architectural style *XPage* following the name of the original framework. This architectural style has been field tested on a number of industrial projects and has proven to be very efficient in terms of both the development times and ease of maintenance.

Before discussing the XPage style in details, we are going to review the related work in Section 2. Section 3 outlines the software engineering challenges that we faced when creating and evolving the XPage framework and architectural style. Section 4 presents the architectural style and its components and connectors. Section 6 discusses the guidelines that we followed and the lessons that we learned in the course of designing and deploying XPage. Section 5 provides cases studies from real-world systems developed based on this style. Finally, Section 7 wraps up the paper with conclusions.

2 Related Work

A complete framework for developing data-driven applications should address a wide array of concerns, from providing efficient data storage and retrieval mechanisms, to handling complex user interactions in the presentation and view layer. To our best knowledge, no other architectural style has been proposed to support development of data-driven systems to this extent. However, there are some solutions proposed by the database research community which focus on related problems. Many of these solutions focus on the idea of developing an entire application based on an *ER* (Entity Relationship) model [4], or an augmented version of it.

The most notable example in this category is WebML [2, 3]. WebML is a product which provides a model-based development environment with a database-oriented view for creating hypertext-based applications. The core of the application is created with a "structural model" which outlines the data model. Special-purpose data-aware "units" or "operations" are devised for data presentation or modification tasks. A program is created by associating these special units with the objects defined in structural model. The result is an appropriate user interface element which can carry out the desired task. A "navigation model" is used to establish the links between different pages and content units.

in [12, 13], Vigna proposes a solution based on developing the entire application out of the Extended ER model. In their solution, the cardinality constraints on entity relationships in the ER model are used to infer the appropriate presentation and navigation model of the application. Their software generates SQL statements necessary for creating the tables based on this augmented ER model. The developer is expected to execute the scripts to create the underlying database. User interface forms are also automatically created based on the schema in the ER model. Further customization of the application should be done by directly modifying the generated forms afterwards.

Even though the organization of data-driven applications is mostly influenced by the structure of their underlying data repositories, ER models lack the expressive power required to specify the structure and behavior of an entire application for a variety of reasons. In any data-driven system, a key factor in designing the navigation and presentation model is the predefined flow of information according to the underlying business processes. This information is not captured in the ER model. Having a flexible software development framework requires devising mechanisms for specifying the business logic and view organization of an application independently of its underlying data model. Another undesirable side-effect of using ER models is that since RDBMS systems are not directly based on ER models, these ER-based methods have to assume responsibility for creating and managing the relational database as well. However, this is inflexible and counter-productive, since in many real-world situations there are database experts who prefer to design and fine tune the database independently. The need for working with legacy databases poses a similar problem.

Recently, we have seen the introduction of some industrial software development frameworks whose goal is to enable web developers to create data-driven applications more efficiently and rapidly. Examples of these frameworks include Ruby on Rails and CakePHP. The core of these frameworks is based on the concept of *Active Records*, which provide a two-way mapping between object classes and database tables. Any instantiation or modification of objects of these types is directly reflected on their associated tables. Foreign key relationships are exposed in Active Records by linking object attributes to their related objects. To capture the logic of an application, these frameworks encourage developers to write "controllers" which are service entry points for user defined operations on the data. However, there is no support for higher-level components of any form in the view layer of these frameworks. The "scaffolding" technique can be used to rapidly create the view layer code out of the structure of an Active Record. However, the produced artifact is low-level code and the relationship between this code and the original Active Record can be lost with subsequent modifications to the either artifacts. Another major inefficiency with these frameworks is that the standard mechanism for retrieving data from Active

Records involves traversing them row by row to reach individual data objects. This suggests a low-level programming style, which for most data-driven scenarios can be totally abandoned for a more high-level view of the "entire data set".

None of these solutions address the software engineering side of the problem. Although these approaches facilitate implementation of data-driven applications, their lack of an explicit architectural design makes it difficult to analyze these solutions with regard to issues of interest to the software engineering community. Moreover, since the relationship between implementation-level constructs and the software components and connectors is not clear, one can not easily determine their potential for reuse across different domains. Nor can one try to formalize a process for designing, implementing, and maintaining the software components needed in these solutions. To the contrary, XPage's explicit architectural style makes it easier to understand, adopt, and extend.

Another class of solutions which have been extensively employed in creating data-driven systems are various middle-ware technologies such as Enterprise Java Beans [1]. They offer standardized interfaces for accessing and manipulating data sources. These are accompanied with basic services such as concurrency, distribution, security and component naming and registry services. These technologies can provide the platform for handling the data storage and retrieval tasks in data-driven systems, and thereby answer one side of the problem. However, they do not offer specialized solutions for the view layer. Moreover, these technologies do not suggest any particular configuration for their surrounding system. The assumption is that developers use "glue" code to instantiate, utilize and maintain these object whenever necessary. Despite being flexible, this is less in line with the spirit of software architectures, which advocate reuse by formalizing exemplification of good engineering solutions.

In [9, 10] Chris et al. present the OODT reference architecture, which is an architecture-based solution for locating data sources and aggregating data from distributed data providers. Their proposed software components and connectors provide the services of data source registry, identification, and querying on top of the industrial middle-ware technologies. Although OODT components and connectors can be employed for creating data-driven systems, like middle-ware technologies OODT does not offer any solution for the view layer of these systems, mainly because its focus is on a different problem. Abstracting and modeling the interactions in the view layer of these applications is much more complex than modeling the data operations, which more or less exhibit a straight-forward input output model. Lastly, like middle-ware technologies, OODT's solution is "programmer-intensive" [9] as it does not employ a high-level ADL.

3 Software Engineering Challenges

In this section, we discuss some of the key software engineering challenges that we needed to address when designing XPage. Some of these challenges are not specific to data-driven systems. However, the specific requirements and characteristics of systems in this domain necessitate analyzing these issues from a narrower perspective. When developing XPage, we also needed to understand and analyze some important trade-offs in design of the components and the overall architecture of our solution. Our answers to these issues had a direct impact on the current design of the XPage.

Efficiency vs. Flexibility – Creating a generic solution and then claiming that it can be applied to an entire class of problems is indeed a brave statement. Proper analysis of any solution would begin with scrutinizing the problem that it is supposed to address. However, in case of a generic solution, not all usage scenarios can be predicted. This makes the evaluation of such solutions difficult, since we can not make statements about the requirements of the scenarios that we have not encountered yet.

However, in general we could notice a trade-off between efficiency and flexibility. By efficiency we mean the effectiveness of the architectural style in codifying a data-driven solution with the least amount of effort. In other words, the more efficient solutions are, the less they need to be told how to realize the set of given requirements for a target system, as they can automatically *infer* how to do so. On the other hand, the more a solution is able to infer the behavior of an application, the less flexible it becomes, since those assumptions limit its ability to be applied to systems with new and different requirements. By flexibility we mean the extent to which different types of applications with different requirements can be developed using the framework architecture.

The trade-off comes from the observation that the more work a given component does automatically, the more likely it is that it does something which is not suitable for some future requirement. Given the great similarities among data-driven applications, our goal was to increase the efficiency of the architectural style with as little effect on its flexibility as possible.

Component/Connector Granularity – Consider any particular implementation of the components and connectors of an architectural style. Be it a modular or an object oriented implementation, one can extract its underlying implementation-level constructs and present them as the building blocks of an alternative architectural style, which would have *finer-grained* components. Obviously, this new alternative style can be applied to the same problems as

the original style, however, its produced solutions are going to be less efficient. On the other hand, if certain architectural configurations reappear in many parts of applications created based on some original architectural style, it can be taken as a sign that the style is not very efficient and that introduction of additional *coarser-grained* components/connectors to encapsulate the repeatedly used configuration can increase the efficiency of the architectural style in modeling those applications.

Since not all possible scenarios for using an architectural style can be identified at the time of its design, determining the optimum component granularity becomes a non-trivial task. In Section 4 we explain how XPage solves this problem using a two-level multi-granular architecture.

Inclusion vs. Extension – In object oriented analysis, two standard idioms of *include* and *extend* are well known for specifying reuse in artifacts like Use Cases. We can extend these concepts to study relationships between the *services* offered by components or connectors in software architectures. More specifically, include can be used to describe a generalization relationship denoting inclusion of the service offered by another component. It is like the *including* component invokes the service of the *included* component as part of its own service. Extend can be used to describe a generalization relationship denoting the extension of the service of another component. The *extending* component accomplishes this by virtually inserting additional action sequences into the service of the *extended* component.

Choosing which relationship to use has an influence on the flexibility and efficiency of the architecture. The crucial difference between these two types of reuse is that the extended component should have already devised the necessary *extension points* to allow insertion of additional action sequences, but the included component does not need to make any such provisions. Using include relationships results in solutions with greater flexibility, as the included service does not impose any specific architecture on its surrounding system. However, such kind of reuse necessitates using a lot of glue code to drive the included components and services. If many of the system's scenarios are similar, this creates repetitive code. Middle-ware technologies usually exhibit the include relationship with their surrounding system. On the other hand, using extend relationships can result in more efficient architectures, as this allows for packing more functionality in the extensible services without compromising flexibility. Event-based architectures are a good example of architectures which use the extend relationship. The extend relationship is more in line with the spirit of software architectures, however the drawback to using extend relationships is that it increases the complexity and alleviates the problem of architectural mismatch [6].

We faced this challenge especially when designing the components and connectors of the view layer. In order to make the view components as efficient as possible, we preferred to have them directly interact with the end-user and the environement. Otherwise, we would have to use repetitive code on every web page to load and instantiate components, and to pass user interface events to the components and to prepare their output for proper presentation to the end-user. This implied that the components needed to take care of the entire cycle of their operation from receiving end-user input from the environment, to processing or retrieving the data, to presenting the results back to the end user. For this, we had to use the extend relationships to make it possible for developers to intervene with this default operation cycle and change the default component behaviors when necessary.

Native Support for User Interface Development – Intensive user interaction is a characteristic of data-driven systems. On the other hand, user interface design and implementation is a complex task and takes up a considerable part of the design and development cycle. However, in data-driven systems user interactions are primarily concerned with primitive operations on the underlying data. Because of the great similarity between all parts of data-driven systems, using low-level user interface toolkits like Java Swing in an ineffective solution. An efficient architectural style for this domain is expected to offer native support for specifying user interfaces. Designing these user interface services as extensible services in the components and connectors of the architectural style makes it possible to streamline development of many parts of data-driven systems. Extension points can be used to specialize the basic user interface services for more complex scenarios.

High-level Programming Support – Due to the great similarity different parts of data-driven systems, using low-level programming languages results in repetitive code for realizing simple and recurrent requirements like loading and storing data. Not only conventional programming languages are inefficient for this task, but also they cause serious problems when it comes to maintaining or evolving the system. We ideally expect the components and connectors of the architectural style to be programmable using high-level configuration files. Separating components' implementation code from the configurations of individual instances has the advantage that that architecture can evolve without modifying the applications based on it. In addition, reusing the implementation and the configuration is more straightforward once they are completely separate from each other.

Programmer-Friendly Development Method – From the programmer's point of view, it is better for related elements to be stored closer to each other to facilitate writing and maintaining the code. On the other hand, in order to avoid repeating common structure and behavior in com-

ponents, we need to "pull up" common functionality and "pushing down" specialized functionality in the object types hierarchy. However, this results in separation of otherwise related defining attributes in components, which is directly reflected in the component configuration files. Gathering related items together in configuration files without sacrificing the optimality of component designs is a non-trivial challenge.

Performance – Architectural styles often introduce additional layers of abstraction into the design of a system. This may have a negative impact on the performance of application which follow the style. On the other hand, using configurable high-level components and connectors incurs an instantiation and configuration overhead which might degrade the system's responsiveness.

Complexity of Real-World Systems – Real world applications inadvertently need to deal with issues and requirements including security, user management, authentication, and internationalization. A successful architectural style for data-driven systems must offer first-class components and connectors to support these requirements.

4 The Architectural Style

In this section, we introduce the XPage architectural style and its accompanying development framework. Enumerating all the components and connectors in the architecture is beyond the scope of this publication. However, we try to present the key components and connectors that constitute the core of the architecture. First we provide an overview of the style and its key characteristics. We then proceed to introduce some individual components and connectors.

4.1 Overview

4.1.1 Overall Architecture

An XPage application is comprised of a set of interconnected View Pages. A View Page can be seen as an abstraction of a web page, or a desktop form. Each View Page, in turn, contains one or more View Forms. The View Forms are data-aware components which can directly interact with the end user. XPage offers different types of View Forms for common data acquisition, manipulations, and presentation tasks. Each View Form is connected to one or more Data Sources. A Data Source abstracts the data model of the underlying data source or destination. Data Sources, in turn, are associated with Data Adapters, which are connectors whose function is to provide a consistent interface for different data repositories available to the application.

XPage components and connectors rely on a predefined launch and configuration protocol for their operation. Upon receiving a request for a specific View Page from the end user, a Coordinator connector locates the corresponding XML file, and instantiates the View Page component. This process is repeated for the View Forms in the loaded View Page and for any other components and connectors referenced in them, including the data layer components. Once the component and connector hierarchy is loaded, a sequence of events are propagated in the hierarchy starting at the root View Page component. Some of the most important events are *load, init, register, process_read, process_write, commit*. Depending on their function, each component and connector may do a different task upon receiving these events. However most components read data from data repositories upon receiving *process_read*, and write back data upon receiving *process_write*.

4.1.2 Component and Connector Granularity

The components and connectors of XPage are divided into two distinct groups, based on their granularity. The *coarse-grained* components and connectors, are first-level players in the architecture of an application. They bundle considerable amount o functionality to make them capable of handling significant data-driven responsibilities in a data-driven application. However the XPage style defines these coarse-grained components and connectors in terms of a number of common fine-grained components and connectors. The fine-grained components and connectors are generalizations of the common structural and behavioral constituents of the coarse-grained components. However, these fine-grained components can not exist independently. For example, given just enough information on the structure of an underlying repository, a coarse-grained View Form can present a data entry form to the user, receive and validate the user values, and store the values in a data source, all without any intervention from the developer. However, this is actually accomplished by cooperation of the constituent fine-grained components in the View Form. Modeling the higher-level components in terms of their constituents increases the efficiency of the style, both by streamlining the development of its framework, and by allowing the developers to reuse common code between different View Forms of the same application.

4.1.3 Communication

On another axis, components and connectors are divided into two categories of view layer and data layer. View Pages, View Forms, and their associated components and connectors comprise the view layer of an XPage application. Data Sources, Data Adapters, and other fine-grained components associated with them constitute the data layer of an XPage application. The communication mechanism

of components and connectors is different for the two layers. In the data layer, components communicate with direct synchronous messages. In the view layer, communications take place by sending asynchronous messages. All view layer messages go through a universal connector, called the Transroute connector. The Transroute can locate view layer components and connectors using their addresses, which are assigned based on their place in the hierarchy of the application. Every component's address is used as a prefix for its internal components and connectors. Upon receiving the *register* event, the addresses in the component hierarchy is registered in the Transroute, so that subsequent messages can be delivered.

Data-driven applications show a set of frequently required interactions on the user interface. For instance, in many cases components rely on receiving foreign key parameters to determine what data to display. In case of a data entry form, a foreign key parameter must be received to establish a relationship between a new data row and other rows in the database. Other frequent scenarios are requesting particular sort orders or filters on presented data. As another example, in most cases data entry forms present foreign key attributes as drop down boxes which show entries from the referred table in the relationship. The XPage provides first-class components and connectors to support efficient implementation of these features. All of the above mentioned examples are handled in the XPage style through sending and receiving special Message objects between components on the same page, or on different pages.

Messages can be private or public. Private Messages have a particular recipient address, while global messages carry parameter-like values and are available to the entire component hierarchy. XPage offers a Message type hierarchy describing all the different messages that can be exchanged between the view layer components. All user interface events and all inter-component communications take place by sending Message objects. Some components and connectors have specific Message Maps. A Message Map registers a component's interest in receiving a specific private Message whenever some particular global Message becomes available. The private and global messages allow components to implement "push-", or "pull-"style communications, which are both handy in data-driven systems.

4.1.4 The XPage Language

All the coarse-grained and fine-grained components of XPage are defined using an XML-based domain-specific language. A set of all such XML files is enough to describe an XPage application. The enables the developers to reuse the XML files for the common components and connectors, and thereby increase the development efficiency and reduce maintenance costs. The XPage framework is able to virtually execute the XML files that describe an XPage application. In the earlier versions of the XPage framework, the XML files were translated on the fly into an interpreted language. In the current implementation, XML files are loaded to instantiate and initialize the components and connectors upon each request. However, an object caching mechanism is employed to increase the performance of the application.

The existing development framework for XPage is designed specifically for the web platform. Although we have not yet deployed XPage in a desktop environment, the view components and the messaging mechanisms are all abstracted away from the web platform in order to make it possible to deploy the style in other environments.

4.1.5 Extensibility

Clearly, complex applications have requirements which can not be satisfied with the generic functionalities embedded in XPage components and connectors. All components and connectors feature a number of *extension points*, whose purpose is to allow developers to customize the behavior of components and connectors. These extension points are usually associated with the predefined events. There are two extension points for every event. For example, corresponding to the *process_write* event, there are two extension points called *before_process_write, after_process_write*. Developers can plug in customized code in these extension points using the native language of the platform on which XPage is deployed. These extensions can directly control the behavior of the components. For example, a data entry form can use the *after_process_write* extension to make additional validations and then prevent the component from storing the data by canceling the *process_write* event.

In the following sections, we describe the XPage components and connectors in more detail.

4.2 Data Layer

Data Adapter – This coarse-grained connector is used to abstract away the heterogeneous interfaces of data repositories in a data-driven system. Whether the data repository is an RDBMS, an XML file, or a gateway to a remote web service, appropriate Data Adapters make them available to the application with a consistent interface for data retrieval and manipulation tasks. For data retrieval, an abstract query object is passed to the Data Adapter. This query object contains information about the source of data, the requested filters on the data, and the requested sort order. It is up to the Data Adapter to translate the query into a language understandable by the underlying data provider. For instance, it can be translated into a SQL query, an XPath query, or a web service invocation message. Likewise, for manipulation of

data, the Data Adapter is provided with a target, the criteria for locating the affected data items, the requested operation, and a set of new or modified values. Data Adapters also offer transactional services.

Data Source – This coarse-grained component is used to elevate the flat data representation that Data Adapters deal with to a hierarchical object model suitable for complicated interactions that view layer components need. Like Data Adapters, Data Sources provide data retrieval and manipulation interfaces, however in a more structured manner. As part of its specification, a Data Source specifies a query object which determines the source and destination of the data, a set of *Data Attributes* which correspond to the columns in the query, and a Data Adapter which can interface with the actual data repository. Data Sources can also enforce various integrity constraints.

Data Attribute – In its simplest form, this fine-grained component corresponds to a column in a query definition. Any entity which works with a Data Source, must configure the Data Attributes of that Data Source before requesting any data operation. For example, for filtering the retrieved data based on some attribute, a filter message must be sent to the corresponding Data Attribute of the Data Source. For requesting the Data Source to store a new data row, all the Data Attributes are initialized with values and then the Data Source is asked to store those values as a new data row. Likewise for retrieving data, the data retrieval request is sent to the Data Source, and then the values are collected from the Data Attributes.

The Data Attributes components have a type hierarchy which determines their features and capabilities. Some of the important Data Attribute types are Base Data Attribute, Primary Key Data Attribute, and Foreign Key Data Attribute. Most of the time, Data Source queries involve many joined tables besides a main base table, and only the attributes from the base table are defined of type Base Data Attribute. Primary Key and Foreign Key Data Attributes are required for data operations like create, update and delete. They let the Data Source know which portion of the data values should be used to locate the affected data items. They also allow the Data Source to enforce various integrity constraints. The type of a Data Attribute also determines to which requests that Data Attributes can respond. For example, it determines whether the attribute is updateable or searchable. More complex Data Attributes like the Seeded Data Attribute can interface with auxiliary Data Sources to automatically calculate derived and aggregated values.

Domain – This fine-grained component is used to help guarantee the validity of data handled by Data Sources. If a Data Attribute is associated with a particular Domain component, all requests for writing to or reading from that Data Attribute pass through the associated Domain component for validation. Each Domain component provides two services of *read*, and *write*. In addition to checking validity of values, these two services can also convert between internal and view-level representations data values. For example, thousands separators can be automatically added and removed for numbers. The XPage framework offers a set of domains for common data types like dates and numbers. A generic regex-based Domain component makes it possible to define new Domains using regular expressions for validation. More complex Domains can be defined by extending the interface of the Domain components using native code.

4.3 View Layer

View Page – An XPage application is implemented as a set of View Pages, which are coarse-grained components. View Pages contain View Forms, which are data-aware components, and other independent components which only aid with the presentation of the page.

View Form – View Forms are coarse-grained data-aware components with specialized functionalities. Currently there are five types of View Forms, corresponding to the five primitive operations on data: Create Form, Read Form, Update Form, Delete Form, and Search Form. These components offer common services which are usually required in implementing a data-driven application. For instance a Read Form can display the data in an underlying Data Source in a grid-like format. It can paginate the data rows, and automatically change the sort order if the end user clicks on one of the grid columns. It also allows users to download its data set as a CSV file. A Search Form can ask the end user for filter criteria and pass the filters to an associated Read Form, thereby reusing the functionality of the Read Form for displaying search results. A Create Form can automatically validate user input, and warn the user if required entries are missing or invalid. A Delete Form can consult the integrity constraints in the data model to check for validity of a delete operation before attempting it, and show appropriate error messages to the end user. All these services are provided by default. However, since many of the above services are required in more than one component, and also to allow developers to exercise more control over the behavior of components, the XPage style defines all the View Forms in terms of a number of common fine-grained components which are discussed in the following items.

View Attribute – These fine-grained components correspond to the individual data-aware element types on a View Form. Most View Attributes are directly connected to a Data Attribute from some Data Source. The type of collaboration between the View Attribute and the Data Attribute depends in part on the surrounding View Form. For example, for a Read Form, after the Read Form sends a query to the Data Source, it asks the Data Source to load a single row

of data into its Data Attributes. Then the View Attributes are instructed to collect the data values from the Data Attributes. The collected values are stored in View Cells for presentation on the GUI. This process goes on for all rows in the query result. View Attributes also respond to some specific Messages. For example, upon receiving a Filter Message, the View Attribute routes a request for filtering the underlying data based on the received values to the Data Attribute. As another example, on a Read Form, View Attributes appear as the header of data columns. A user click on these column headers sends a Sort Message to the View Attribute, which is likewise routed to the associated Data Source.

View Cell – These are fine-grained components which represent the individual data-aware elements. View Cells are not definable by the developer. Rather they are produced at run time during the operation of the View Forms. A View Cell may represent a data entry field on a form, or an individual value in a grid of displayed data. The View Cells have links to the original View Attributes that instantiated them.

View Input – These are fine-grained components which represent the user interface widgets. Each View Attribute is associated with some View Input, which determines how that View Attribute is presented on the user interface. In most cases the type of the View Attribute, or its underlying Data Attribute, can automatically suggest the type of the View Input, so that developers do not need to specify them.

One of the most important View Inputs in the XPage style is the Drop Down List. By default, all View Attributes linked to Foreign Key Data Attributes are represented as Drop Down Lists on the user interface. The Drop Down List is automatically given the Data Source component matching the target entity of the foreign key relationship. This enables the Drop Down List to show the end user corresponding values for from the referenced entity.

View Filter – View Filter are connectors whose purpose is to receive filter requests via special Filter Message objects and relay the filter to the associated Data Source. While every View Attribute is capable of filtering the underlying Data Source, View Filters are useful when the View Form relies on receiving the filter message for its operation. For example, an Update Form usually needs to work with an individual data row, and refuses to operate if it does not receive a proper Filter Message.

View Link – View Links are fine-grained connectors which connect different View Pages in an XPage application. All inter-page messages are sent through the View Links. Depending on the containing View Form, a View Link may appear as a hyperlink taking a user from one page to the other, or as a form submit button. In either case, each View Link carries a number of Message objects. A typical scenario is sending a foreign key as a global Message to a target View Filter in another page. Another one is sending user input on a data entry form as a set of Input Messages.

View Template – Each visible XPage component or connector is associated with some View Template for its type. View Templates determine how the element is displayed on the user interface. Standard View Templates are provided in the XPage framework for all objects. This enables developers to customize the presentation of an individual component without having to provide custom presentations for its contained elements.

4.4 Support Components

Enumerating all the component and connectors in XPage is beyond the scope of this paper. However we can quickly mention some of the remaining key players in the XPage style. Coordinator connectors are responsible for locating and loading other components during the process of loading the component hierarchy of a View Page. Based on the type of the requested components, Coordinators can decide intelligently whether they can reuse previously loaded components or not. This is particularly important for sharing Data Adapters for ensuring transactional integrity, as in order to successfully roll back transactions, all related operations on a View Page must be handled by a single Data Adapter. Coordinators also allow for deploying the XPage style in a distributed environment, as they can potentially return stubs for remote components and connectors. Security in the system is enforced by defining access rights on components and connectors. An Authentication component provides the user logon facility. A Navigation component can dynamically create navigational menus by matching the set of all available components against the access rights of the current user. An i18n component allows developers to provide internationalization support with little effort. All captions, comments, examples, and other textual information on the view layer components are translated through the i18n component before being displayed.

5 Case Studies

In this section, we introduce some real-world implementations of the XPage style.

5.1 The Lepidoptera Collection at FLMNH

Florida Museum of Natural History (FLMNH) at University of Florida is home to many research collections and databases. The Museum possesses over 24 million objects (specimens, artifacts, materials) in its various collections. Over the years, different software technologies have been used to develop data management systems for these collections. Although these systems have different implemen-

tations, they are quite similar in terms of their functions and their interfaces, mainly due to the nearly identical features of all Museum collection databases. For instance, most of these systems offer mechanisms for managing specimens/artifacts data, locality information, taxonomic hierarchies, and loans.

In summer 2006, XPage was used to develop the Lepidoptera (butterflies and moths) collection's data management system. Deploying XPage at FLMNH provided us with a unique opportunity to compare the performance of XPage with other software technologies that were used for developing similar applications in the Museum. We compared the Lepidoptera application with the Ceramics collection application developed in ASP, and the Archeology collection application developed in Ruby on Rails. Table 1 shows the data that we have gathered on the development of these three web-based applications.

Table 1. test

Application	Ceramic (ASP)	Archeology (Rails)	Lepidoptera (XPage)
Tables	37	32	32
Unique tables	26	21	20
Fields	112	477	202
Relationships	35	22	28
UI forms	157	86	152
Total code	36966 loc 1324 KB	7174 loc 285 KB	4857 loc 183 KB
Presentation	None None	4101 loc 186 KB	300 loc 14 KB
Configuration	None None	269 loc 10 KB	4494 loc 166 KB
Development time	5 months	5.5 months	1.5 months

Despite the common features in all three applications, each system had to deal with specific requirements. For example, the Zooarcheology application was ajax enabled, the Ceramics application offered complimentary interfaces for a public web site, and the Lepidoptera application allowed users to import and export data. In data-driven applications, the size of the underlying database is a good indication on the size of the problem. So for each application we counted the number of tables, the number of fields, and the number of relationships. Since usually the application store similar entities in multiple tables for organizational or implementation reasons, we also counted the number of unique table types. In order to account for variations in the amount of functionality offered, we also counted the number of user interface forms/pages in all three applications. UI Forms were counted multiple times if they offered more than a single operation on data. To capture the size of the solution, we measured the size of the produced programs. Whenever possible, we separated the user interface code for the presentation of the application as well as the configuration code.

The most notable fact in this comparison is that in the Lepidoptera application, 92% of total code is just XML configuration files. In fact, this application used only 63 lines of low-level PHP code in the UI forms. This is the main reason behind its shorter development time. The bulk of the implementation of XPage took place in less than seven days. The remaining time was used for requirements analysis, database design, and incremental updates due to subsequent requests. In the Archeology application, more than 50% of the time was spent on UI development. Although this application offered an ajax-enabled UI, it used a consistent and similar UI architecture for all forms, which naturally results in repetitive code. For the Ceramics application, we couldn't separate the presentation code or the configuration code from the other parts of the system, as everything is mixed up in the ASP files. Aside from the shorter development time, the XPage application has the advantage that developers work with XML configuration files which are more readable, more reliable, and more maintainable than low level code used in the alternative solutions.

The relatively higher number of fields for the Archeology application is due to maintaining four nearly identical copies of the main artifacts table (45 fields) for each subcollection. In addition, another table with 22 columns is replicated 8 times for implementation purposes. It is while in the other two applications, the replicated tables have 1-2 fields only.

5.2 Squash

In [5], Esfahbod et al. use XPage to implement a web-based front-end for configuring an organizational gateway giving users controlled access to a set of internally administered infobases. The front-end allows end users to define available infobases together with a hierarchy of organizational users in different levels and groups. Access rights can be assigned to individual users or to organizational levels. An offline cron job processes the information in the database and generates suitable configuration files for a squid web proxy which acts as the gateway to the actual infobases.

Despite unfamiliarity of this group with the XPage architecture and its framework, it took their three member team only six person-days to implement Squash. At that time, only the Create Forms offered Drop Down Lists, and this form of user input was not available in Search Forms. Since they needed such a feature on their search forms, and they did not want to bother modifying the XPage framework, they used the extension points as a workaround. They used a Create Form in place of the Search Form, however they override the *before_process_write* extension to cancel the create operation and instead route the user input mes-

sages to the Search Form for being processed as filter criteria.

5.3 BibIS

In 2006, XPage was used to develop a Bibliographic database for the Database Research Group at the Department of Computer Science at University of Florida. The core functionality of BibIS is to manage research publications. It allows end users to manage publication types, enter publication information, and optionally upload actual article files. In addition to browsing and searching the publications, BiBIS allows users to produce BibTeX entries for any set of selected publications. These requirements could be implemented using nothing but default XPage components and connectors.

However, the interesting requirement in BibIS was that all the publication attributes and publication type attributes needed to be dynamically definable by the end user. Again, we used the extension points to satisfy this requirement. An *empty* Data Source was defined with no Data Attributes for the publication entity. In the *before_load* extension point of this Data Source, we included custom code to load an auxiliary Data Source on the table containing publication meta-data and initialize the publication Data Source according to the user defined attributes. This allowed for containing the complexity of this solution. All View Forms which worked with the publication entity were developed as if the publication table was a static table. Other extension points were used to update the structure of the actual publication table as the end users added/removed the meta-data on publication attributes.

5.4 Ringtone Vending Website

Our last case study is from the deployment of XPage on an Internet web site for selling cell phone ringtones and logos. This system worked on three geographically distributed servers. A catalog server on content providers site offered web services for getting information on available content for sale. The web server presented the catalog to the Internet users and accepted orders. Received orders were sent to a GSM server, which communicated with the content provider to get the ringtone and then send it to end user's cell phone.

Since the standard View Templates produce a simple look and feel for the components, we used custom View Templates to produce a suitable presentation for a public web site. Although there is no intrinsic support for web services in XPage, we used the extension points on two virtual Data Sources to provide the connection from the web server to the catalog server and the GSM server. The first virtual Data Source made a web service request to the catalog server upon a read request to get the catalog information. The second virtual Data Source was used as if it was saving user orders. However, instead it activated local scripts which sent order parameters to the remote GSM server.

A common theme that is seen in all the above experiences is a need for extensibility in an architectural style. All these systems had requirements which were not predicted when XPage was designed. This confirms the importance of extensibility as a key requirement for any generic software engineering solution.

6 Guidelines and Lessons Learned

In this section, we present some guiding principles that we either followed, or later learned that we *should have followed* from the beginning in the process of designing the XPage style. Some of these decisions were made after trial and errors in the gradual process of evolving the XPage's accompanied framework. It would be very difficult to quantify some of these statements, as it is impossible to change only one dimension of such a solution and compare it with other alternatives. Nevertheless, we list them here as our guiding principles.

Separation of Concerns – The early prototypes for XPage were based on View Forms only. Apparently, a single data entity appears on multiple View Forms, for example when it needs to be updated, browsed, or created. So, we had to repeat the textual attributes like caption and comments on all View Forms containing that data entity. This led us to designing additional constructs to hold those common values. The structure of these additional constructs turned out to very much resemble the current Data Sources. A similar pattern led us to develop the View Templates in an effort to reuse components presentations despite their different inner workings. The current architecture of XPage more or less follows the Model-View-Controller (MVC) [8] pattern. In turns out that we could have shortened the number of iterations by strictly following the MVC pattern from the beginning.

Bottom-up Development of the Style – Although no one single instance can represent the requirements of all systems in a class, having one instance is better than having none. The original XPage framework was developed for a single system and was later evolved for developing each new system. Many of the design challenges do not show up in hypothetical "hello world" style applications. The laboratory generated sample applications tend to focus on exercising all the features offered in the solution. It is while real-world applications show non-uniform distributions for feature requests, which is a key factor for optimizing the solution's efficiency by a taking appropriate design decisions.

Repeating Database Integrity Constraints in the Application – We needed to properly identify invalid data op-

erations before attempting them on the data repository to give the end user better error messages. Also, many view layer components could automatically configure themselves if they could have more information on the data attributes that they presented. For example, a foreign key attribute is automatically represented as a drop down list, and an Update Form can automatically configure View Filters for all the primary key attributes in its underlying Data Source. Adding "hints" here and there to guide these feature gradually materialized into a copy of all integrity constraints in the design of Data Sources. These constraints also have the benefit that they can protect against developer errors. For example, a Delete Form by default refuses to attempt a delete operation if its embedded View Filters have not received proper messages, which prevents the component from executing a "delete all".

Multi-Granular Components and Connectors – Describing the fine-grained components and connectors which operate inside the coarse-grained components has had numerous advantages. These include increasing the developer's understanding of the style, streamlining reuse in the implementation of the architectural style's development framework, allowing for reuse in the application domain through copying partial component configurations, decreasing copy-and-pasting errors due to the similarity between definition languages of similar constituents, and finally giving developers the ability to choose the right tool for the right job – use coarse-grained solutions for typical scenarios, and configure fine-grained solutions for the few complex requirements.

Making Common Scenarios Easy and Complex Scenarios Possible – Based on the experience with deploying XPage in real-world systems, we have tried to tailor the default behavior of View Forms in favor of the typical simple data entry and presentation tasks. So, instead of relying on developer for correctly configuring the appropriate fine-grained components to achieve the desired function, we have configured the default View Forms such that in most cases no customization is needed. On the other hand, developers can exercise a lot of control over the View Forms by manually adding or removing fine-grained components and connectors, and by plugging in extra code in the extension points.

Programmer-Oriented Development – One of the interesting features of XPage language is that it allows developers to have some degree of flexibility in deciding the organization of component configurations in XML files. Wherever a containing component needs to include another component, the developer has the freedom to provide the configuration of the contained component inline or use a reference to an existing configuration file. For example, the description of the Data Source of an Update Form can be provided entirely inside the `<datasource>` tag, or this tag can use the `ref` attribute to reference an external XML file for the Data Source. This is easily made possible by the Coordinator connectors, as all requests for loading contained components is handed over to Coordinators. Although for large enough systems, we ideally expect to use external references to avoid repetition of configurations, this flexibility enables developers to start writing "lousy" configuration files when prototyping or for small applications, and then later enhance the organization if needed. Another aspect of XPage's programmer-oriented guiding principles is that it uses the native language of the target platform for defining the extension points, instead of inventing a new high-level language.

Handling Conventions and Defaults – Another axis of flexibility is in defining structural and behavioral attributes in related components. For example, a View Attribute should have a caption, which appears next to it on the user interface. The caption can be defined as an XML attribute in the View Form's configuration file. However, XPage has a convention that View Attribute will receive the caption from their associate Data Attribute, if they are not explicitly given one. Likewise, the Data Attributes can use a caption defined in the associated Domains. So although logically the caption belongs on the View Attribute, developers can define it in lower levels to increase reuse. As another example, XPage has the convention that a View Attribute is automatically associated with the Data Attribute which has the same name, if it is not associated explicitly. These conventions allow developers to lower the size of the configuration files in addition to simplifying the maintenance of the application.

7 Conclusions

We have presented the XPage architectural style, and discussed the software engineering challenges that we faced during its design. Data-driven systems are relatively unstudied by software researchers due to their imagined low complexity. Our successful experience with XPage shows that it is possible to streamline many activities involved in design and development of data-driven systems. However, we believe data-driven systems have much more potential for reuse and we are looking forward to seeing more research devoted to discovering techniques and methods for exploiting this great potential.

References

[1] Enterprise java beans. *http://java.sun.com/ejb/*.

[2] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (webml): a modeling language for designing web sites. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications netowrking*,

pages 137–157, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[3] S. Ceri, P. Fraternali, and M. Matera. Conceptual modeling of data-intensive web applications. *IEEE Internet Computing*, 6(4):20–30, 2002.

[4] P. P.-S. S. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[5] B. Esfahbod and H. S. Allah. Squash: Design and implementation of a large scale http gateway and masqurader. *Internet draft: http://behdad.org /download /Publications /squashdoc /squash.pdf*, 2003.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 179–185, New York, NY, USA, 1995. ACM.

[7] D. Garlan and M. Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 2:1–39, 1993.

[8] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[9] C. A. Mattmann, D. J. Crichton, J. S. Hughes, S. C. Kelly, and P. M. Ramirez. Software architecture for large-scale, distributed, data-intensive systems. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, page 255, Washington, DC, USA, 2004. IEEE Computer Society.

[10] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. A software architecture-based framework for highly distributed and data intensive scientific applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 721–730, New York, NY, USA, 2006. ACM.

[11] D. E. Perry and A. L. Wolf. Foundation for the study of software architecture. *Software Engineering Notes*, 17(2):40–52, 1992.

[12] S. Vigna. Erw: Entities and relationships on the web. *Poster Proc. of Eleventh International World Wide Web Conference*, 2002.

[13] S. Vigna. Automatic generation of content management systems from eer-based specifications. *ase*, 00:259, 2003.